Linköping University Electronic Press

Book Chapter

2D Shape Rendering by Distance Fields

Stefan Gustavson

Part of: OpenGL Insights: OpenGL, OpenGL ES, and WebGL community experiences, ed. Patrick Cozzi, Christophe Riccio ISBN: 978-1-4398-9376-0

Available at: Linköping University Electronic Press
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-91558

Contents

1 2D Shape Rendering by Distance Fields 1
- 1.1 Introduction . . . 1
- 1.2 Method Overview . . . 2
- 1.3 Better Distance Fields . . . 4
- 1.4 Distance Textures . . . 4
- 1.5 Hardware Accelerated Distance Transform . . . 5
- 1.6 Fragment Rendering . . . 6
- 1.7 Special Effects . . . 6
- 1.8 Performance . . . 7
- 1.9 Shortcomings . . . 9
- 1.10 Conclusion . . . 9
- Bibliography . . . 10

Index 11

2D Shape Rendering by Distance Fields

Stefan Gustavson

1.1 Introduction

Every now and then, an idea comes along that seems destined to change the way certain things are done in computer graphics, but for some reason it is very slow to catch on with would-be users. This is the case with an idea presented in 2007 by Chris Green of Valve Software, in a SIGGRAPH course chapter entitled "Improved Alpha-Tested Magnification for Vector Textures and Special Effects" [Green 07]. Whether the slow and sparse adoption is due to an obscure title, the choice of publication venue, a lack of understanding from readers, lack of source code, or the shortcomings of Green's original implementation, this chapter is an attempt to fix that.

The term *vector textures* refers to 2D surface patterns built from distinct shapes with crisp, generally curved boundaries between two regions: foreground and background. Many surface patterns in the real world look like this, for example printed and painted text, logos, and decals. Alpha masks for blending between two more complex surface appearances may also have crisp boundaries: bricks and mortar, water puddles on asphalt, cracks in paint or plaster, mud splatter on a car. For decades, real-time computer graphics has been long plagued by an inability to accurately render sharp surface features up close, as demonstrated in Figure 1.1. Magnification without interpolation creates jaggy, pixelated edges, and bilinear interpolation gives a blurry appearance. A common method for alpha masks is to perform thresholding after interpolation. This maintains a crisp edge, but it is wobbly and distorted, and the pixelated nature of the underlying data is apparent.

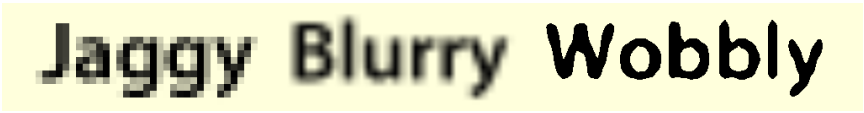

Figure 1.1. Up close, high-contrast edges in texture images become jaggy, blurry or wobbly.

Shape rendering by the method described here solves the problem in an elegant and GPU-friendly way, and it does not require re-thinking the production pipeline for texture creation. All it takes is some insight into what can be done. This chapter aims at providing that insight. First, we present the principles of the method, and explain what it is good for. Following that, we provide a summary of recent research on how to make better distance fields from regular artwork, removing Green's original requirement for special high-resolution 1-bit alpha images. Last, we present concrete shader code in GLSL to perform this kind of rendering, comment on its performance and shortcomings, and point to trade-offs between speed and quality.

1.2 Method Overview

Generally speaking, a crisp boundary cannot be sampled and reconstructed properly using standard texture images. Texel sampling inherently assumes that the pattern is *band limited*, i.e. that it does not vary too rapidly and does not have too small details to be represented by a smooth interpolated reconstruction from the texel samples. If we keep one of these constraints, that the pattern must not contain too small details, but want the transitions between background and foreground to be crisp, formally representing an infinite gradient, we can let a shader program perform thresholding by a *step function* and let the texels represent a smoothly varying function on which to apply the step. A suitable smooth function for this purpose is a *distance field*.

A typical distance field is shown in Figure 1.2. Here, texels do not represent a color, but the distance to the nearest contour, with positive values on one side of the contour and negative values on the other. An unsigned distance field, having distance values only outside the contour, is useful, but for flexibility and proper anti-aliasing it is highly preferable to have a signed distance field with distance values both inside and outside the contour. The contour is then a *level set* of the distance field: all points with distance value equal to zero. Thresholding the distance function at zero will generate the crisp 2D shape. Details smaller than a single texel can not be represented, but the boundary between background and foreground can be made infinitely sharp, and because the texture data is smoothly varying and can be closely approximated as a linear ramp at most points, it will behave nicely under both magnification and minification using ordinary bilinear interpolation.

Thresholding by a step function will alias badly, so it is desirable to instead use a linear ramp or a `smoothstep` function, with the transition region extending across approximately one fragment (one pixel sample) in

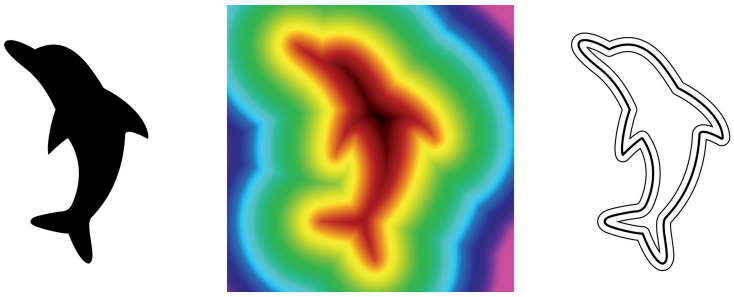

Figure 1.2. A 2D shape (left), its smoothly varying distance field shown in a rainbow color map (middle) and three level sets (right) showing the original outline (thick line) and inwards and outwards displaced outlines (thin lines).

the rendered output. Proper anti-aliasing is often overlooked, so Listing 1.1 gives the source code for an anisotropic anti-aliasing step function. Using the built-in GLSL function `fwidth()` may be faster, but it computes the length of the gradient slightly wrong as $|\frac{\partial F}{\partial x}|+|\frac{\partial F}{\partial y}|$ instead of $\sqrt{\frac{\partial F}{\partial x}^2 + \frac{\partial F}{\partial y}^2}$. Using ± 0.7 instead of ± 0.5 for the thresholds compensates for the fact that `smoothstep()` is smooth at its endpoints and has a steeper maximum slope than a linear ramp.

```
// 'threshold' is constant, 'distance' is smoothly varying
float aastep(float threshold, float distance) {
  float afwidth = 0.7 * length(vec2(dFdx(distance), dFdy(distance)));
  return smoothstep(threshold-afwidth, threshold+afwidth, distance);
}
```

Listing 1.1. Anisotropic anti-aliased step function.

Because the gradient of a distance field has a constant magnitude except at localized discontinuities, the *skeleton points*, gradient computation is straightforward and robust. The gradient can be stored with the distance field using a multi-channel (RGB) texture format, but it can also be accurately and efficiently estimated by the automatic derivatives `dFdx()` and `dFdy()` in the fragment shader. Thus, it is not necessary to sample the texture at several points. By carefully computing the gradient projection to screen space, an accurate, anisotropic analytical antialiasing of the edge can be performed with little extra effort.

1.3 Better Distance Fields

In digital image processing, distance fields have been a recurring theme since the 1970's. Various *distance transform* methods have been proposed, whereby a binary (1-bit) image is transformed to an image where each pixel represents the distance to the nearest transition between foreground and background. Two problems with previously published methods are that they operate on binary images, and that they compute distance as a vector from the center of each foreground or background pixel to the center of the closest pixel of the opposite type. This only allows for distances that are of the form $\sqrt{i^2 + j^2}$, where i and j are both integers, and the measure of distance is not consistent with the distance to the *edge* between foreground and background. These two restrictions have recently been lifted [Gustavson and Strand 11]. The new *anti-aliased Euclidean distance transform* is a straightforward extension of traditional Euclidean distance transform algorithms, and for the purpose of 2D shape rendering, it is a much better fit than previous methods. It takes as its input an anti-aliased, area-sampled image of a shape, it computes the distance to the closest point on the underlying edge of the shape, and it allows fractional distances with arbitrary precision, limited only by the anti-aliasing accuracy of the input image. The article cited contains the full description of the algorithm, with source code for an example implementation. The demo code for this chapter contains a similar implementation, adapted for stand-alone use as a texture preprocessing tool.

1.4 Distance Textures

The fractional distance values from the anti-aliased distance transform need to be supplied as a texture image to OpenGL. An 8-bit format is not quite enough to represent both the range and the precision required for good quality shapes, but if texture bandwidth is limited, it can be enough. More suitable formats are, of course, the single channel `float` or `half` texture formats, but a 16-bit integer format with a fixed-point interpretation to provide enough range and precision will also do the job nicely.

For maximum compatibility with less capable platforms such as WebGL and OpenGL ES, we have chosen a slightly more cumbersome method for the demo code for this chapter: we store a 16-bit fixed-point value with 8 bits of signed integer range and 8 bits of fractional precision as the R and G channels of a traditional 8-bit RGB texture. This leaves room for also having the original anti-aliased image in the B channel, which is convenient for the demo and allows for an easy fallback shader in case the shape rendering turns out to be too taxing for some particularly weak GPU.

The disadvantage is that OpenGL's built-in bilinear texture interpolation incorrectly interpolates the integer and fractional 8-bit values separately, so we need to use nearest-neighbor sampling, look up four neighbors explicitly, reconstruct the distance values from the R and G channels, and perform bilinear interpolation by explicit shader code. This adds to the complexity of the shader program. Four nearest-neighbor texture lookups constitute the same memory reads as a single bilinear lookup, but most current hardware has built-in bilinear filtering that is faster than doing four explicit texture lookups and interpolation in shader code. (The OpenGL extension `GL_ARB_texture_gather`, where available, goes some way towards addressing this problem.)

A bonus advantage of our approach using dual 8-bit channels is that we work around a problem with reduced precision in the built-in bilinear texture interpolation. We are no longer interpolating colors to create a blurry image, but computing the location of a crisp edge, and that requires better precision than what current (2011) GPUs provide natively. Moving the interpolation to shader code guarantees an adequate accuracy for the interpolation.

1.5 Hardware Accelerated Distance Transform

In some situations where a distance field might be useful, it can be impractical or impossible to pre-compute it. In such cases, a distance transform can be performed on the fly using a multi-pass rendering and GLSL. An algorithm suitable for the kind of parallel processing that can be performed by a GPU was originally invented in 1979 and published as little more than a footnote in [Danielsson 80] under the name *parallel Euclidean distance transform*. It was recently independently reinvented under the name *jump flooding* and implemented on GPU hardware [Rong and Tan 06]. A variant that accepts anti-aliased input images and outputs fractional distances according to [Gustavson and Strand 11] is included in the accompanying demos and source code for this chapter. The jump flooding algorithm is a complicated image processing operation that requires several iterative passes over the image, but on a modern GPU, a reasonably sized distance field can be computed in a matter of milliseconds. The significant speedup compared to a pure CPU implementation could be useful even for off-line computation of distance fields.

1.6 Fragment Rendering

The best way of explaining how to render the 2D shape is probably to show the GLSL fragment shader with proper comments. See Listing 1.2. The shader listed here assumes the distance field is stored as a single channel floating point texture. As mentioned above, the interactive demo instead uses a slightly more cumbersone 8-bit RGB texture format for maximum compatibility. A minimal shader relying on the potentially problematic but faster built-in texture and anti-aliasing functionality in GLSL is presented in Listing 1.3. It is very simple and very fast, but on current GPUs, interpolation artifacts appear even at moderate magnification. A final shape rendering is demonstrated in Figure 1.3, along with the anti-aliased image used to generate the distance field.

Figure 1.3. *Left:* A low resolution, anti-aliased bitmap. *Right:* Shapes rendered using a distance field generated from that bitmap.

1.7 Special Effects

The distance field representation allows for many kinds of operations to be performed on the shape, like thinning or fattening of features, bleed or glow effects and noise-like disturbances to add small scale detail to the outline. These operations are readily performed in the fragment shader and can be animated both per-frame and per-fragment. The distance field representation is a versatile image-based component for more general procedural textures. Figure 1.4 presents a few examples of special effects, and their corresponding shader code is shown in Listing 1.4. For brevity, the example code does not perform proper anti-aliasing. Details on how to implement the `noise()` function can be found in Chapter **??**.

```
// Distance map 2D shape texturing, Stefan Gustavson 2011.
// A re-implementation of Green's method, using a single
// channel high precision distance map and explicit texel
// interpolation. This code is in the public domain.

#version 120
uniform sampler2D disttex; // Single-channel distance field
uniform float texw, texh;  // Texture width and height (texels)
varying float oneu, onev;  // 1/texw and 1/texh from vertex shader
varying vec2 st;           // Texture coords from vertex shader

void main( void )
{
  vec2 uv = st * vec2(texw, texh); // Scale to texture rect coords
  vec2 uv00 = floor(uv - vec2(0.5)); // Lower left of lower left texel
  vec2 uvlerp = uv - uv00 - vec2(0.5); // Texel-local blends [0,1]

  // Perform explicit texture interpolation of distance value D.
  // If hardware interpolation is OK, use D = texture2D(disttex, st).

  // Center st00 on lower left texel and rescale to [0,1] for lookup
  vec2 st00 = (uv00  + vec2(0.5)) * vec2(oneu, onev);
  // Sample distance D from the centers of the four closest texels
  float D00 = texture2D(disttex, st00).r;
  float D10 = texture2D(disttex, st00 + vec2(0.5*oneu, 0.0)).r;
  float D01 = texture2D(disttex, st00 + vec2(0.0, 0.5*onev)).r;
  float D11 = texture2D(disttex,st00+vec2(0.5*oneu,0.5*onev)).r;
  vec2 D00_10 = vec2(D00, D10);
  vec2 D01_11 = vec2(D01, D11);
  vec2 D0_1 = mix(D00_10, D01_11, uvlerp.y);  // Interpolate along v
  float D = mix(D0_1.x, D0_1.y, uvlerp.x);    // Interpolate along u

  // Perform anisotropic analytic antialiasing
  float aastep = 0.7 * length(vec2(dFdx(D), dFdy(D)));
  // 'pattern' is 1 where D>0, 0 where D<0, with proper AA around D=0.
  float pattern = smoothstep(-aastep, aastep, D);
  gl_FragColor = vec4(vec3(pattern), 1.0);
}
```

Listing 1.2. Fragment shader for shape rendering.

Figure 1.4. Shader special effects using plain distance fields as input.

1.8 Performance

We benchmarked this shape rendering method on a number of current and not-so-current GPUs, and instead of losing ourselves in details with a table, we summarize the results very briefly.

```
#version 120
uniform sampler2D disttex; // Single-channel distance field
varying vec2 st;           // Texture coords from vertex shader

void main( void )
{
  float  D = texture2D(disttex, st).
  float aastep = 0.5 * fwidth(D);
  float pattern = smoothstep(-aastep, aastep, D);
  gl_FragColor = vec4(vec3(pattern), 1.0);
}
```

Listing 1.3. Minimal shader, using built-in texture interpolation and AA.

```
// Glow effect
  float inside = 1.0 - smoothstep(-2.0, 2.0, D);
  float glow = 1.0 - smoothstep(0.0, 20.0, D);
  vec3 insidecolor = vec3(1.0, 1.0, 0.0);
  vec3 glowcolor = vec3(1.0, 0.3, 0.0);
  vec3 fragcolor = mix(glow * glowcolor, insidecolor, inside);
  gl_FragColor = vec4(fragcolor, 1.0);

// Pulsate effect
  D = D - 2.0 + 2.0 * sin(st.s * 10.0);
  vec3 fragcolor = vec3(smoothstep(-0.5, 0.5, D));
  gl_FragColor = vec4(fragcolor, 1.0);

// Squiggle effect
  D = D + 2.0 * noise(20.0*st);
  vec3 fragcolor = vec3(1.0 - smoothstep(-2.0, -1.0, D)
   + smoothstep(1.0, 2.0, D));
  gl_FragColor = vec4(fragcolor, 1.0);
```

Listing 1.4. Shader code for the special effects in Figure 1.4.

The speed of this method on a modern GPU with adequate texture bandwidth is almost on par with plain, bilinear interpolated texturing. Using the shader in Listing 1.3, it is just as fast, but the higher quality interpolation of Listing 1.2 is slightly slower. Exactly how much slower depends strongly on the available texture bandwidth and ALU resources in the GPU. With some trade-off in quality under extreme magnifications, single channel 8-bit distance data can be used, but 16-bit data comes at a reasonable cost. Proper anti-aliasing requires local derivatives of the distance function, but on the hardware level this is implemented as simple inter-fragment differences with very little overhead.

In short, performance should not be a problem with this method. Where

speed is of utmost importance, decals and alpha masks could in fact be made smaller with this method than with traditional alpha masking. This saves texture memory and bandwidth and can speed up rendering without sacrificing quality.

1.9 Shortcomings

Even though the shapes rendered by distance fields have crisp edges, a sampled and interpolated distance field is unable to perfectly represent the true distance to an arbitrary contour. Where the original underlying edge has strong curvature or a corner, the rendered edge will deviate slightly from the true edge position. The deviations are small, only fractions of a texel in size, but some detail may be lost or distorted. Most notably, sharp corners will be shaved off somewhat, and the character of such distortions will depend on how each particular corner aligns with the texel grid.

Also, narrow shapes that are less than two texels wide cannot be accurately represented by a distance field, and if such features are present in the original artwork, they will be distorted in the rendering. To avoid this, some care needs to be taken when designing the artwork and when deciding on the resolution of the anti-aliased image from which to generate the distance field. Opposite edges of a thin feature should not pass through the same texel, nor through two adjacent texels. (This limitation is present also in traditional alpha interpolation.) Both these artifacts are demonstrated by Figure 1.5, which is a screenshot from the demo software for this chapter.

1.10 Conclusion

A complete cross-platform demo with full source code for texture creation and rendering is freely available through `http://www.openglinsights.com`.

This chapter and its accompanying example code should contain enough information to start using distance field textures in OpenGL projects where appropriate. Compared to [Green 07], we provide a much improved distance transform method taken from recent research and give example implementations with full source code for both texture generation and rendering. We also present shader code for fast and accurate analytic anti-aliasing, which is important for the kind of high-frequency detail represented by a crisp edge.

While distance fields certainly do not solve every problem with rendering shapes with crisp edges, they do solve some problems very well, for example text, decals and alpha-masked transparency for silhouettes and

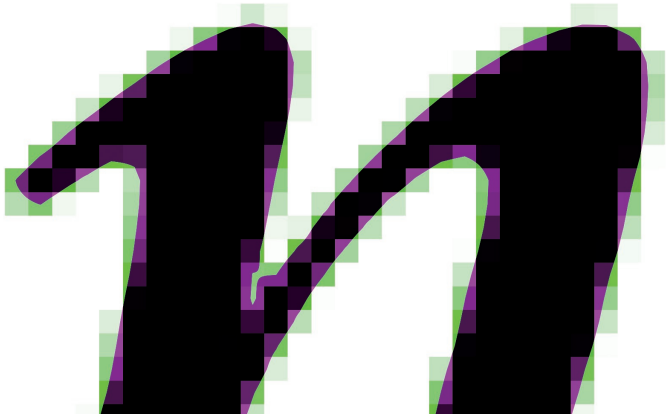

Figure 1.5. Rendering defects in extreme magnification. The black and white shape is overlaid with the grayscale source image pixels in purple and green. For this particularly problematic italic lowercase "n", the left edge of the leftmost feature is slightly rounded off, and the narrow white region in the middle is distorted where two opposite edges cross a single texel.

holes. Furthermore, the method does not require significantly more or fundamentally different operations than regular texture images, neither for shader programming nor for the creation of texture assets. It is our hope that this method will find more widespread use. It certainly deserves it.

Bibliography

[Danielsson 80] Per-Erik Danielsson. "Euclidean distance mapping." *Computer Graphics and Image Processing* 14 (1980), 227–248.

[Green 07] Chris Green. "Improved Alpha-Tested Magnification for Vector Textures and Special Effects." In *Siggraph07 Course on Advanced Real-Time Rendering in 3D Graphics and Games, Course 28*, pp. 9–18, 2007.

[Gustavson and Strand 11] Stefan Gustavson and Robin Strand. "Anti-Aliased Euclidean distance transform." *Pattern Recognition Letters* 32:2 (2011), 252–257.

[Rong and Tan 06] Guodong Rong and Tiow-Seng Tan. "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform." In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, pp. 109–116, 2006.

Index

band limited, 2

distance field, 2
distance transform, 4

jump flooding, 5

level set, 2

noise, 7

step function, 2

vector textures, 1